Software Defined Networking Basics

Anupama Potluri
School of Computer and Information Sciences
University of Hyderabad

Software Defined Networking (SDN) is considered as a paradigm shift in how networking is done. SDN was triggered by many different developments in the industry: [2]

1. Availability of good switch chips that make fast switching possible.
2. Big data centers.
3. Frustration with big vendors that leads to tie-in to their equipment and prevents fast innovation and higher costs.
4. Virtualization which is not very well handled by standard networking.

1 Motivation for SDN

There are many factors tht motivated the development of SDN. A lot was driven by data center requirements. Academic principles also drove the development of SDN. Primarily the lack of principles in networking and its poverty as an academic discipline were also motivating factors. We review these factors that motivated SDN development.

Manageability: Most of the networking protocols and therefore, the configuration on different routers/switches deal with lower level namespaces such as IP addresses, MAC addresses etc. In today's networks, it makes sense to have higher level constructs such as user names, host names for policies. This level of abstraction is not currently possible. For e.g., we cannot (in a campus network) easily determine and limit the amount of download/student per month. Such configuration and tracking require a different type of software currently not supported in routers. We need a software that maps the abstract names to lower level namespaces and allows network policies to be specified in higher level abstractions.

Control Plane Modularity: Control plane modularity (or the lack thereof) is another motivating factor for SDN. For e.g., different control plane protocols such as OSPF/BGP/IS-IS need the network topology information to determine forwarding paths for packets. Each of them implements their own distribution mechanism that is used to collect the topology information. When we look at the code of OSPF in the routers, 95% is to do with the part that gathers the topology and 5% is the actual Dijkstra algorithm [2]. By allowing the topology to be collected once and reused by all the control programs that need the topology, we get much better modularity.

Virtualization: Virtualization led to increased mobility of user machines. State information associated with each user machine - for e.g., security policies, QoS policies - had to migrate to the new physical switch where the user VM has migrated to. But, there is no mechanism to configure switches automatically as the VMs migrate to them. These switches are typically configured using either CLI/SNMP Manager/HTTP by a network administrator. This simply kills the mobility of user machines as far as networking is concerned. This led to the development of the programmable data plane and a virtualized switch that allows network migration to happen. For e.g., Open vSwitch (OVS) [6, 7, 8] is primarily a virtual switch that allows migration of network state, better programmability (manageability), inter-VM/intra-PM traffic to be within the PM itself instead of hair-pin switching and so on as explained in Section 6.

Programmable and Generalized Data Plane: As pointed out above, the ability to handle the policies associated with mobile user VMs was a major requirement in large data centers. OpenFlow [5, 3, 4] is the protocol for this purpose primarily. It allows switches to be configured (programmed) for different action items based on match of some packet headers. The packet header fields matched may consist of L2, L3, L4 and even L7 headers and an action corresponding to such a matching item may be used. At this point, OpenFlow is still not at the stage where it is completely generalized and is tightly-coupled to the existing set of protocols and their header contents. However, the hope is that, in future, it will become a very general set of matching rules.

Innovation: Once the various functions are decoupled, as in a routing algorithm only worried about how to determine paths over a given topology, innovations are likely to happen faster. It requires far lesser code and works on well-established APIs that allow visibility to the topology. A programmable data plane leads to separation of production data from experimental protocols. This allows new protocols to be tested in real networks without disrupting regular traffic. This makes it easier to deploy new protocols and leads to faster innovation.

Networking Principles: Networking has so far been just a bag of protocols [1] and no principles have been developed to handle the control plane. Unlike other "systems" disciplines such as OS, DBMS, networking has not matured well. It is not modular as far as control plane is concerned. It does not have any abstractions which leads to replication of functionality and complex code. SDN is about defining these abstractions. There are primarily three abstractions that define SDN:

1. Distributed state as an abstraction
2. Specification abstraction
3. Forwarding abstraction

In the rest of this monograph, we will discuss these abstractions and end with two software products that help achieve these abstractions.

2 Distributed State as an abstraction

The control plane of a router is quite complex as of today since there is no modularity in the way it is built. The control plane deals with the state distributed in various elements of the network. Every router maintains state information about itself and its link status and its neighbours – their reachability and on which link they are connected – that is used to compute paths for flows. The fact that every routing protocol consists of a distributed solution to construct the topology violates basic tenets of modularity. Just as the data plane layering separates the various functionalities and ensures that there is no replication of the same functionality across layers, we need to build this layering in the control plane also.

In short, there is a *logically* centralized **controller process** that collects the distributed state from all the elements of the network. It provides APIs for other programs to view and control the state for various purposes. In other words, the distributed state is abstracted away from the control programs which need not worry about how to construct the state. They need not deal with the many issues of failures in communication while propagating the state information, need not worry about maintaining the state consistent with the current situation etc. Instead, they can assume that the controller will take care of all of these. This is analogous to the way the operating systems were modularized in Unix. Unix's design lead to abstraction (hiding) of hardware details so that only small components of the OS deal with the hardware. If an OS needs to be ported to a different hardware, it is only these parts that need to be handled. A similar layering and abstraction are needed for networking to become more portable, flexible and innovate with new applications.

Network Operating System (NOS): This brings us to the concept of the **Network Operating System** (NOS) that provides interfaces for different programs to access and manipulate the networking elements. Just as disks, ethernet cards etc. are manufactured by different vendors and have different characteristics but these details are hidden from application programs, the networking elements such as switches, routers and servers can be from any vendor and can be hardware or software. These details and their implementation are hidden from the control programs that actually do the job required for the network operators. These are the applications of networking. The network policies need to be "compiled" into *switch independent* instructions. These instructions are relayed to the switches which actually implement the translation of these instructions into specific commands of that switch. This is like the device drivers, which are the only component of OS that deal with the actual hardware. A compiler translates from a higher level language to machine level instructions. All these abstractions help an application programmer develop the "application" without worrying about how to implement it on a particular hardware. Control programs in networking should be able to specify applications that control the elements to implement policies, get traffic information at a particular instant of time, use the traffic to dynamically modify the control state of the network and so on. This should not mean that the control program is aware of how each element operates.

Commoditisation of switches: The interesting side-effect of this separation of control and data planes is commoditisation of switches *a la* the commoditisation of servers. The trend in server virtualization was to replace custom-made costly servers, that led to vendor lock-in, with off-the-shelf commodity servers. Whenever an increase in the capacity of the servers was required, instead of increasing the RAM, CPU cores etc., a new server is added to handle the

load. Commoditization leads to cheaper products as well as makes one component replaceable by another leading to greater flexibility. A similar trend is to be seen in networking if the cost of networks has to come down and allow for greater flexibility in managing networks.

Let us take an example: if a router goes down today in a network, all the networks reachable through this router are inaccessible. If there exist alternate paths, the control plane has to discover them and set them up in the data plane before they become available. If downtime of this sort is not acceptable, there are many standard methods of high availability used. There can be redundant line cards, redundant routers themselves maintaining state information through checkpointing to seamlessly switch over in case of a failure. However, these are highly expensive solutions. If, on the other hand, the switches are not complicated and all they do is to install forwarding rules in the SRAM/TCAM and forward based on the match of these rules, then, switches can be commoditised. Redundant switches are no longer a costly operation. This is horizontal scaling in switching analogous to horizontal scaling in servers.

The switches can be made extremely simple as stated above if they do only the data plane operations which are extremely simple. The control plane (and management plane) logic which is the complex part of a router can be completely separated and executed on a completely different system. In fact, most ordinary servers can handle quite a bit of control plane traffic and distribute the computed data plane rules to the switches. These can be also redundant or load-balanced to prevent single point of failure and/or congestion. Thus, the router moves from being a tighly coupled system to a distributed system.

3 Forwarding Abstraction

A forwarding abstraction that is highly flexible is essential for the control plane. It should be simple and generalized such as in MPLS/ATM. This hides the various vendor specific hardware details from the control plane. OpenFlow [5] is one such forwarding abstraction.

There are a set of rules to match with an action associated with them. There are also statistics stored with each entry that give the dynamic state of the network. The rules and the action are determined by the control program that runs on top of the network operating system as per the network policy of the organization.

There are two methods by which the rules are installed in the switches – reactively or proactively. In the former case, there is a default rule installed which forwards all packets that do not match any other rule to the controller. The controller consults the global policies of the enterprise and based on the policy matched, installs the corresponding rule into the switches. For e.g., if the packet is a *telnet* packet, the security policy may be to simply drop the packet. In this case, a rule will be installed to drop the packets arriving on the switch for *telnet*. On the other hand, since the policy is not to allow *telnet* traffic at all in the network, this rule can be proactively installed in all the switches before any packet is received by them. The latter will ensure that there is no latency associated with the processing of the traffic and switching happens at line speed. However, it involves installing rules that may never be used and wasting memory in the switches as well as contribute to a longer lookup time. Thus, a good tradeoff between these two approaches might be a better option.

A typically standard method is to install some default rules that capture DHCP, DNS and other data and send them to the controller. This will ensure that the only process that allocates addresses is the controller. This prevents havoc in the network with multiple DHCP

servers installed in an unauthorized fashion leading to duplicate network and/or host IDs. These services may run as VMs on the controller server handling all the traffic.

4 Specification Abstraction

While the distributed state abstraction helps control programs that deal with the state of the network to be freed from figuring out a mechanism to gather the state, these programs will still need to know the physical topology to control it. The **Specification Abstraction** allows arbitrary virtual network topologies to be defined on top of the actual physical topology. "The control program should express desired behavior rather than figure out how to implement this on the physical topology" [1].

The network operator works on an abstract model of the network and specifies policies or behavior of the network. This is the control program. The network virtualization layer maps the abstract model to the global network view. The NOS programs the actual physical switches. Neither the control programs nor the network virtualization layers need to know about how to program the individual switches for achieving a particular function. This is done by the NOS using OpenFlow. The OpenFlow implementation of a particular vendor's switch translates these primitives into the specific entries in the switch flow tables.

A network administrator must be able to specify that there are N VLANs in their network and the topology they wish to have in their network. This needs to be mapped to the actual physical topology in the data center. Their view may be even simpler – it is just a single switch to which all their VMs are connected. The operator may want to specify that user A cannot talk to user B or that user A is allowed only a certain amount of download. How to translate this to the virtual switches/physical switches/ports, apply these rules on the actual physical topology is not the responsibility of the control program. The control program needs to be provided statistics collected on the ports/switches of the *abstract topology*. The control program then enforces the network policies configured by the network operator on this abstract topology. This needs to be mapped to the specific physical switches and ports by the network virtualization layer. The NOS is then informed of the actions to be taken on these physical network elements. The NOS uses OpenFlow to perform these actions.

In summary, this is the *Network Virtualization* layer that sits between the NOS and the control program. The network virtualization layer uses the NOS API to program the physical switches while maintaining a mapping between the virtual topology seen by a tenant and the actual physical topology in the data center. This shields the vagaries of the physical network from the control programs. Thus, the virtual network of a tenant can migrate from one physical server and switch to another physical server and switch transparently. The control program does not have to deal with such events that may occur due to physical failures. FlowN [13] and Flowvisor [14] are examples of network virtualization software that provide this support on top of NOX [12].

Network virtualization helps to implement different addressing schemes and routing strategies on top of the same physical topology, given data plane (forwarding) abstraction described in Section 3. The abstract model of the network can use a totally different addressing scheme, routing protocols and so on. The control programs for this abstract model will implement these new protocols. Let us take the example of IPv6 in the topology given in Fig. 1. If two IPv6 hosts in this topology had to communicate over the IPv4 infrastructure that connected them, we needed transition mechanisms like ISATAP, 6to4 etc. Consider the same situation

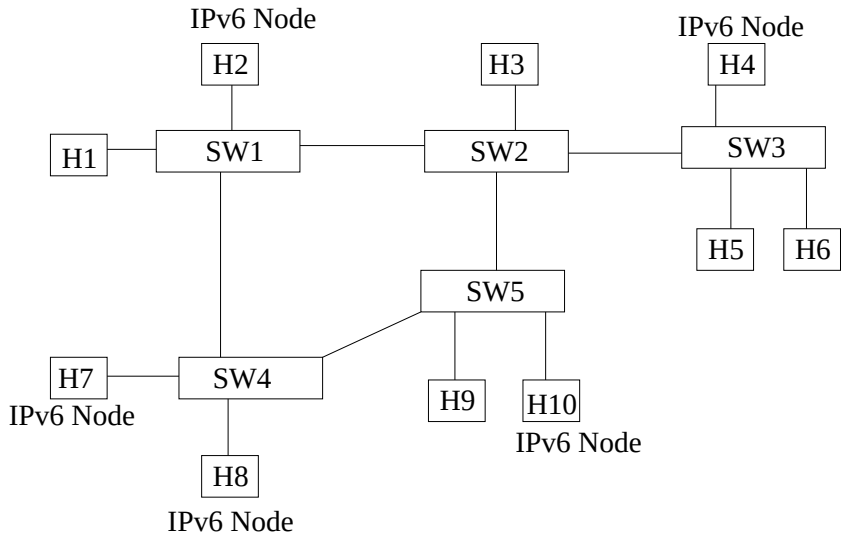

Figure 1: Example Topology with some IPv6 nodes that communicate over IPv4 infrastructure

in the context of SDN. A completely general data plane abstraction allows any bit vector to be matched without any knowledge of L2/L3 protocol header formats. Such switches need to be programmed to match only the bytes 8-24 and 25-40 to identify a particular set of end hosts in IPv6. We can set up the data path through the switches that connect these two hosts. We can actually visualize all the IPv6 hosts to be a single VLAN connected by a single switch. This is the abstract model of this network that an IPv6 experimenter can use. This does not require any modification of the switches on the path or the NOS or the Network Virtualization layer. It requires the developer of a new L3 protocol to simply develop the L3 protocols – both data and control planes – and deploy them as control programs on top of the network virtualization layer.

Thus, the three abstractions help deploy new ideas faster with minimum modifications to the software deployed in the network infrastructure and allow rapid innovation.

5 OpenFlow

OpenFlow [9, 5] was proposed as a standard interface for data plane abstraction. At this point, OpenFlow is still not general enough and is aware of the L2 and L3 protocol headers. In a "Type 0" OpenFlow switch the fileds that are matched are : the input port, VLAN ID, source and destination Ethernet addresses and type, source and destination IP addresses and protocol, source and destination TCP/UDP ports. Each such matching rule has a priority and an action associated with it. The higher priority rules are the ones used when there are multiple rules matching. In addition, counters are also maintained per rule. These return statistics for that flow(s).

An OpenFlow switch consists of three parts:

1. A *Flow Table* having rules to match, action and counters.

2. A *Secure Channel* to communicate with a controller on which commands are sent to the switch and counters are read.

3. The *OpenFlow Protocol* which is the interface between the controller and the switches.

There are two types of OpenFlow switches – dedicated OpenFlow switches and OpenFlow-enabled switches. The former do not support standard L2 and L3 switching. The latter are standard L2/L3 switches that also support OpenFlow for programmability.

Dedicated OpenFlow Switches: The three basic actions associated with each rule that all dedicated OF switches must support are:

1. Forward this flow's packets to a given port (or ports). This allows packets to be forwarded along a given path to the destination.

2. Encapsulate and forward the traffic to a controller using the secure channel to the controller. This is usually done for the intiation packet of a flow so that the controller can consult the policies for this packet and install the required rules in the switches on the path.

3. Drop the flow's packets.

OpenFlow-enabled commercial switches have a fourth rule to augment the three above: forward the traffic through the switch's normal processing.

Other features may also be provided by OpenFlow switches such as re-writing parts of the header (as in the case of NAT), encapsulate in a tunnel, map to a different class of service to provide QoS etc. Switches supporting such features are termed as "Type 1" switches.

6 Open vSwitch

As compute virtualization has spread and has become the *de facto* standard for support of cloud services, networking has not kept pace with it. VMmigration breaks network policies as these policies have to migrate to new switches. However, this still required per-physical switch configuration unlike the *resource pool* configuration in computation. [10]. While earlier there were few VMs/server, today, having 40-100 VMs in a single server is a norm. An IBM Power server has been stress-tested with 300 VMs. This means an entire Class C IPv4 network can be held in a single server! Earlier there was no way to manage the network formed at this software *edge*. The bridge software that came bundled with most operating systems did not have any features to manage the traffic passing through it. Thus, all VM-to-VM traffic within a PM was the most troublesome of the traffic as no policies could be imposed on this traffic nor could it be monitored. Some solutions always passed the traffic through the first physical switch – *hairpin switching* – since the switches were already built for linespeed switching and manageability. However, other solutions, such as *Open vSwitch*, brought the features of standard switches into software. Software switches can take advantage of advanced NIC features such as TCP segmentation offload and checksumming currently available in more advanced NICs. Future NICs may provide ability to offload IPSec processing etc.. Therefore, software switching at the edge can also be at linespeed by taking advantage of these NICs.

Open vSwitch (OVS) provides centralized policy configuration. This allows a set of switches to be treated as a single distributed virtual switch. Policies are applied to virtual interfaces using these switches and migrate along with the VMs to which they were applied.

OVS resides in the hypervisor. It exports interfaces for manipulating the forwarding state and managing configuration state. These interfaces are [11]:

Configuration: A remote process can read and write configuration state as key/value pairs and set up triggers to receive asynchronous events when the configuraiton changes such as a new VM has come up or a VM has gone down etc. It presently provides the ability to turn on port mirroring (as used in spanning tree protocol), apply QoS policies on interfaces, enabled Netflow logging for a VM etc.

Forwarding Path: OVS also provides a way to set up or manipulate the forwarding path, through the OpenFlow protocol.

Connectivity management: OVS provides a local management interface through which the topological configuration can be manipulated. For e.g., many instances of OVS may run on a single physical server creating a complex topology within a single system. It also manages the virtual interface (VIF) connectivity by adding logical ports/VIF as well as physical interface (PIF) connectivity.

OVS implements two components: the "slow path" which is the control logic that includes forwarding logic, MAC learning, load balancing etc. and is implemented in the user space. The "fast path" is the actual forwarding based on a match of packet headers and this is implemented inside the kernel. This separation ensures that the fast path is simple and is the only part dependent on the OS and hence requires porting to a new OS. This helps the overall portability of OVS.

References

[1] The Future of Networking and the Past of the Protocols. Scott Shenker (with Martin Casado, Teemu Koponen, Nick Mckeown and others) Open Networking Summit Keynote, 2011. http://www.opennetsummit.org/archives/apr12/site/talks/shenker-tue.pdf

[2] SDN and Streamlining the Plumbing. Nick Mckeown. COMSNETS-2014.http://comsnets.org/archive/2014/doc/NickMcKeownsSlides.pdf

[3] Software Defined Networking: The New Norm for Networks. https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf

[4] OpenFlow Specification. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf

[5] OpenFlow. http://en.wikipedia.org/wiki/OpenFlow.

[6] Why Open vSwitch? http://git.openvswitch.org/cgi-bin/gitweb.cgi?p=openvswitch;a=blob_plain;f=WHY-OVS;hb=HEAD

[7] Programmable Networks with Open vSwitch. Jesse Gross. LinuxCon. http://events.linuxfoundation.org/sites/events/files/slides/OVS-LinuxCon 2013.pdf

[8] Underneath OpenStack Quantum: Software Defined Networking with Open vSwitch. Thomas Graf, RedHat Inc. April 24, 2013. http://www.cloudcomp.ch/wp-content/uploads/2013/04/OpenStack-Quantum-SDN-with-Open-vSwitch.pdf

[9] OpenFlow: Enabling Innovation in Campus Networks. Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, Jonathan Turner. SIGCOMM Comput. Commun. Rev., pages 69–74, April 2008. http://doi.acm.org/10.1145/1355734.1355746

[10] Virtual Switching in an Era of Advanced Edges. Justin Pettit, Jesse Gross, Ben Pfaff, Martin Casado, Simon Crosby. In Proceededings of the 2nd Workshop on Data Center - Converged and Virtual Ethernet Switching (DC CAVES), September 2010.

[11] Extending Networking into the Virtualization Layer. Ben Pfaff, Justin Pettit, Teemu Koponen, Keith Amidon, Martin Casado, Scott Shenker. HotNets-2009. http://conferences.sigcomm.org/hotnets/2009/papers/hotnets2009-final143.pdf

[12] NOX: Towards an Operating System for Networks. Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martin Casado, Nick McKeown, Scott Shenker. SIGCOMM Comput. Commun. Rev., July 2008. http://doi.acm.org/10.1145/1384609.1384625

[13] Scalable Network Virtualization in Software-Defined Networks. Dmitry Drutskoy and Eric Keller and Jennifer Rexford. IEEE Internet Computing, Vol. 17, No. 2, pages 20-27, 2013. http://doi.ieeecomputersociety.org/10.1109/MIC.2012.144

[14] Can the production network be the testbed? Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown, Guru Parulkar. Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, OSDI-2010. http://dl.acm.org/citation.cfm?id=1924943.1924969